System Services

Cocoa > Interapplication Communication

2002-11-12

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Contents

Figures and Listings

Introduction to System Services

Services give applications an open-ended way to extend each other's functionality by allowing applications to

- provide services to other applications
- access functionality provided by other applications

The shared functionality is accessed through the Services submenu of every application's application menu. An application does not need to know in advance what operations are available; it merely needs to indicate the types of data it uses, and the Services menu makes available the operations that apply to those types of data.

Organization of This Document

This document describes how Mac OS X services work, shows some typical Services menus, and provides instructions on how you can use services in your application. You should read this document if you are an application developer and want to provide your application's services to other applications or make services from other applications available to your application.

Before you read this document, you should be familiar with information property lists. You need to know what they are and how to add properties to a list. See "Information Property Lists" for more information.

For guidelines on naming menu items and designing the interface for a services application, see *Apple Human Interface Guidelines*.

Here are the concepts covered:

- "Services Architecture" (page 9)
- "Items in the Services Menu" (page 13)
- "Services Properties" (page 15)

Here are the tasks covered:

- "Providing a Service" (page 19)
- "Using Services" (page 23)
- "Creating the NSServices Property" (page 27)

Services Architecture

Services allows a user to access the functionality of one application from within another application. An application that provides a service advertises the operations it can perform on a particular type of data—for example, encryption of text, optical character recognition of a bitmapped image, or generating text such as a message of the day. When the user is manipulating that particular type of data in some application, the user can choose the appropriate item in the Services menu to operate on the current data selection (or merely insert new data into the document).

This section discusses how services are processed and describes some sample services.

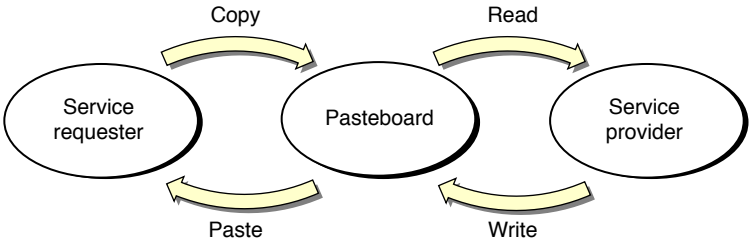
Service Request

Any application that signs up to use services automatically has access to the advertised functionality through its Services menu. An application does not need to know in advance what operations are available; it merely needs to indicate the types of data it uses, and the Services menu makes available the operations that apply to those types of data.

Services are performed by transferring data back and forth between applications through a shared pasteboard. Note that the two applications—service requestor and service provider—are completely separate; they do not run in a shared memory space. The pasteboard holding the data is specific to the service request and does not normally interfere with the standard Copy/Paste pasteboard.

When the user chooses a Services menu item data flows as shown in Figure 1 (page 9). The current selection is copied to a pasteboard which is then passed to the service provider application. If the service provider is not currently running, it is automatically launched. The service provider reads the contents of the pasteboard and operates on it. The service provider writes new data back to the pasteboard and the pasteboard is returned to the original application. The original application then pastes the pasteboard's contents into the document, replacing the current selection, if there is one. The service provider application does not automatically quit at the end of the service request.

Figure 1 Data flow in a service request



Not all services both receive and provide data. Some services only receive data and others only provide data. In these cases only one of the copy and paste steps is performed. Services can thus be divided into two groups:

- Processor. This type of service acts on data. A processor service acts on the current selection and then sends it to the service. For example, if a user selects an email address in a TextEdit document, and then chooses Mail > Mail To from the Services menu, TextEdit copies the person's address to the pasteboard, the Mail application launches, and Mail pastes the address into the Send field of a new email message.
- Provider. This type of service gives data to the calling application. For example, if a user chooses Grab > Screen from the Services menu, the Grab application opens, takes a screen shot, then returns the screen shot (TIFF data in this case) to the calling application. The calling application (such as TextEdit) is responsible for pasting the data into the active document.

A service falls into both categories if it processes the current selection and then provides a replacement value. For example, a text encryption service takes the current text selection, encrypts it, and then returns the encrypted text to the service requester to replace the current selection.

Sample Services

The following few figures show Services in action. Figure 2 (page 10) shows the Services menu from the TextEdit application. Make Sticky is an example of a processor service. The Make Sticky command takes the current selection in the TextEdit document, opens a new Stickies document, and then pastes the selection into the Stickies document. For more convenient use, a keyboard shortcut (Command-Shift-Y) is defined for this service.

Figure 2 Make Sticky is a processor service

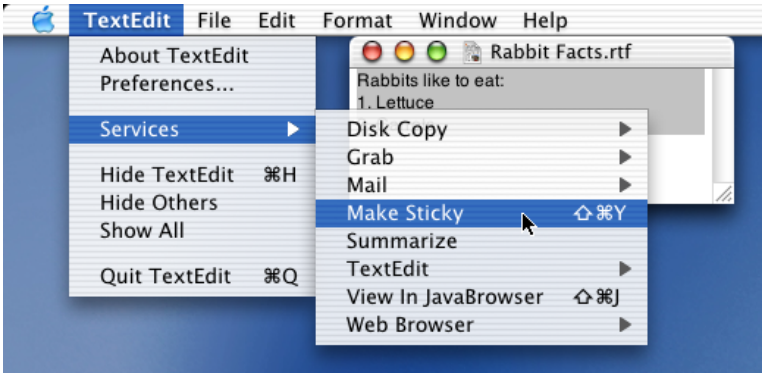
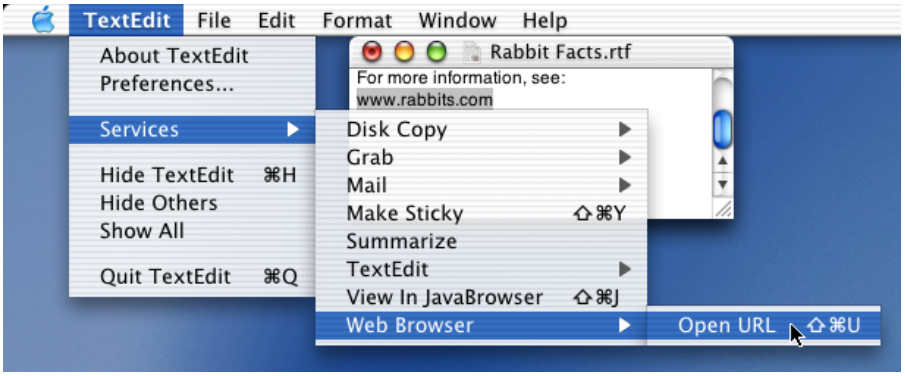


Figure 3 (page 11) shows another example of a processor service. In this case, the Open URL command copies the selected text, launches a Web browser, pastes the selected text into the browser's location field, and then tries to connect to that location.

Figure 3 Open URL is a processor service



Grab is a provider service. Figure 4 (page 11) shows the Wolf Facts document before Grab > Screen is invoked. Figure 5 (page 12) shows the Wolf Facts document after Grab has taken a shot of the current screen and returned the data to the TextEdit application. Recall that it is TextEdit's responsibility to do something with the returned data. In this example, TextEdit simply pastes the TIFF into the current document at the insertion point.

Figure 4 Grab is a provider service

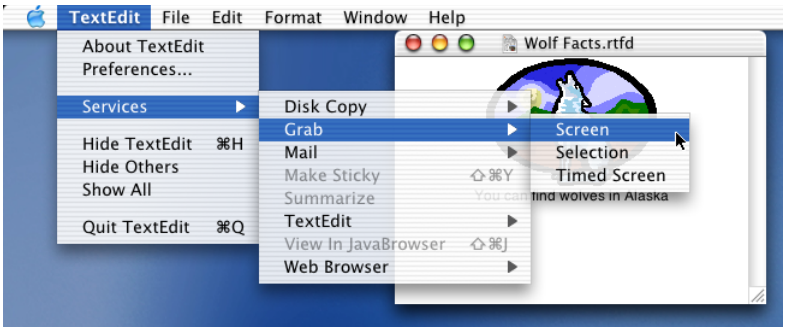
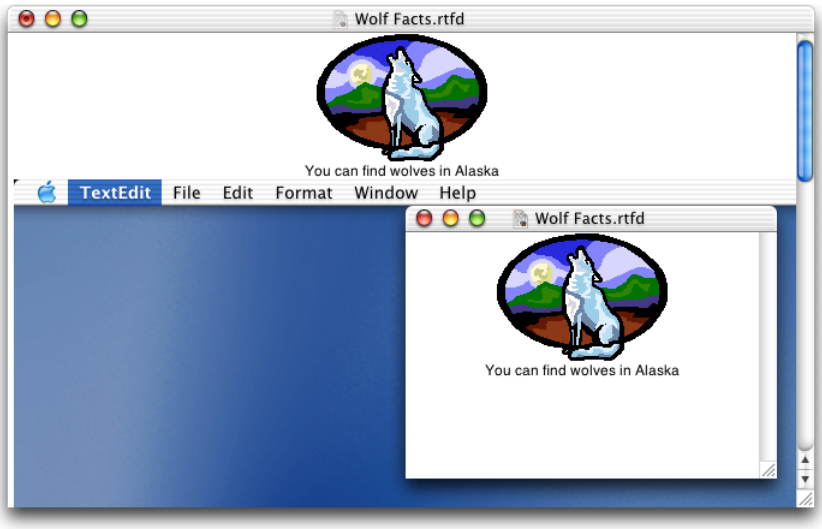


Figure 5 The Wolf Facts document after a screen shot has been inserted



Items in the Services Menu

Applications that provide services are installed in the `Applications` and `Library/Services` folders in any of the four file-system domains—System, Network, Local, and User. (See "File-System Domains" in *System Overview* for details on file-system domains.) The applications' information property lists declare the services the applications provide (see "Services Properties" (page 15)). Mac OS X collects the property list information and uses it to populate the items in the Services menu based on the particular data types supported by each application.

The Services menu is included in the default nib file created by Xcode and Interface Builder for Cocoa applications. If the application's menu is instead created programmatically, you need to designate a Services menu using the NSApplication method `setServicesMenu:`. If an application registers for services (see "Using Services" (page 23)), the appropriate items are automatically available in the Services menu.

The items in the Services menu can be commands or submenus that contain commands. If an application offers only one service, just the service (stated as a command) is listed in the Services menu. For example, the Stickies application offers only one service—making a new Sticky note—so only the command Make Sticky is listed in the Services menu.

If an application offers more than one service, the application's name usually appears in the Services menu, and the services offered by the application appear in a submenu. For example, the Grab application offers three services: taking a screen shot of the entire screen, taking a screen shot of a selected part of the screen, and taking a screen shot of the entire screen after a set amount of time. As you can see in Figure 4 (page 11), Grab is an item in the Services menu that has its own submenu listing the commands that invoke Grab's three services: Screen, Selection, and Timed Screen.

The Services menu is populated when the application launches, but its items are not enabled until the user chooses Services from the application menu. Choosing Services causes the current responder chain to be searched for objects that can provide or receive data of the types used by each service listed in the Services menu. If an object is found that can use a given service, the service's menu item is enabled. Menu items for which no suitable object is found are dimmed, unavailable for the user.

Services Properties

Any application that has one or more services to provide must advertise the type of data its services can handle. Services are advertised through the `NSServices` property of the application's information property list (`Info.plist` file).

Note: The information property list (`Info.plist`) contains key-value pairs that specify the application's properties that are of interest to the Finder and other applications. Although the `Info.plist` is a text file that uses XML (Extensible Markup Language) format, you should not modify the XML directly unless you are very familiar with XML syntax. Instead, use Xcode or the Property List Editor application provided with Mac OS X to modify the `Info.plist` file. You can find more information on property lists in *System Overview*.

Property Definitions

`NSServices` is a property whose value is an array of dictionaries that specifies the services provided by the application. Keys for each dictionary entry, are as follows:

- `NSMessage` indicates the instance method to invoke. Its value is used to construct an Objective-C method of the form *`messageName`*`:userData:error:` or a Java method of the form *`messageName`*`(NSPasteBoard,String)`. This message is sent to the application's service provider object.
- `NSPortName` is the name of the port to which the application should listen for service requests. Its value depends on how the service provider application is registered. In most cases, this is the application name.
- `NSMenuItem` is a dictionary that specifies the text of the Services menu item. The solitary entry has the key `default`. Its string value is the menu item text. You can use a slash to specify a submenu. For example, `Mail/Send Selection` appears in the Services menu as a submenu named Mail with an item named Send Selection. `NSMenuItem` must be unique, as only one is used in the Services menu if there are duplicates.

  To localize the string, create a `ServicesMenu.strings` file for each localization in your bundle with the above `default` menu item string as the lookup key. For example, create a `.strings` file that has `Mail/Send Selection` as the key and the localized text as its value. (See *System Overview* for details on localized strings files.) If a localized string is not found, the `default` text is used.
- `NSKeyEquivalent` is an optional dictionary that specifies the keyboard equivalent that invokes the menu command. Like `NSMenuItem`, the only entry in the dictionary should have the key `default` with a string value that can be localized in the `ServicesMenu.strings` file. The string value must be a single character. The keyboard shortcut is this one character with the Command and Shift key modifiers.

  Use key equivalents sparingly. Remember that your shortcuts are being added to the collection of shortcuts defined by each application as well as defined by the other services. When an application already has a shortcut with that key equivalent, the application's shortcut wins. If multiple services define the same shortcut, which one gets invoked is undefined.

- `NSSendTypes` is an array that contains data type names. Send types are the types sent from the application requesting a service. The NSPasteboard class description lists several common data types. An application that provides a service must specify `NSSendTypes`, `NSReturnTypes`, or both.
- `NSReturnTypes` is an array that contains data type names. Return types are the data types returned to the application requesting a service. The NSPasteboard class description lists several common data types. An application that provides a service must specify `NSSendTypes`, `NSReturnTypes`, or both.
- `NSUserData` is an optional string that contains a value of your choice. You can use this string to customize the behavior of your service. For example, if your application provides several similar services, you can have the same `NSMessage` value for all of them (each service invokes the same method) and use different `NSUserData` values to distinguish between them. This entry is also useful for applications that provide open-ended, or add-on, services.
- `NSTimeout` is an optional numerical string that indicates the number of milliseconds Services should wait for a response from the application providing a service when a response is required. If the wait time exceeds the timeout value, the application aborts the service request and continues without interruption. If you don't specify this entry, the timeout value is 30000 milliseconds (30 seconds).

Add-On Services

You typically define services when you create your application and advertise them in the `Info.plist` file of the application's bundle. The services facility also allows you to advertise services outside of the application bundle, enabling you to create "add-on" services after the fact. This is where the `NSUserData` entry becomes truly useful: You can define a single message in your application that performs actions based on the user data provided, such as running the user data string as a UNIX command or treating it as a special argument in addition to the selected data that gets sent through the pasteboard. To define an add-on service, you create a bundle with a `.service` extension that contains an `Info.plist` file, which in turn contains the add-on service specification. The service specification uses the application's `NSMessage` and `NSPortName` values.

Sample Property List

The `NSServices` property for the Grab application is shown in Figure 1 (page 17) as it appears in the Property List Editor application.

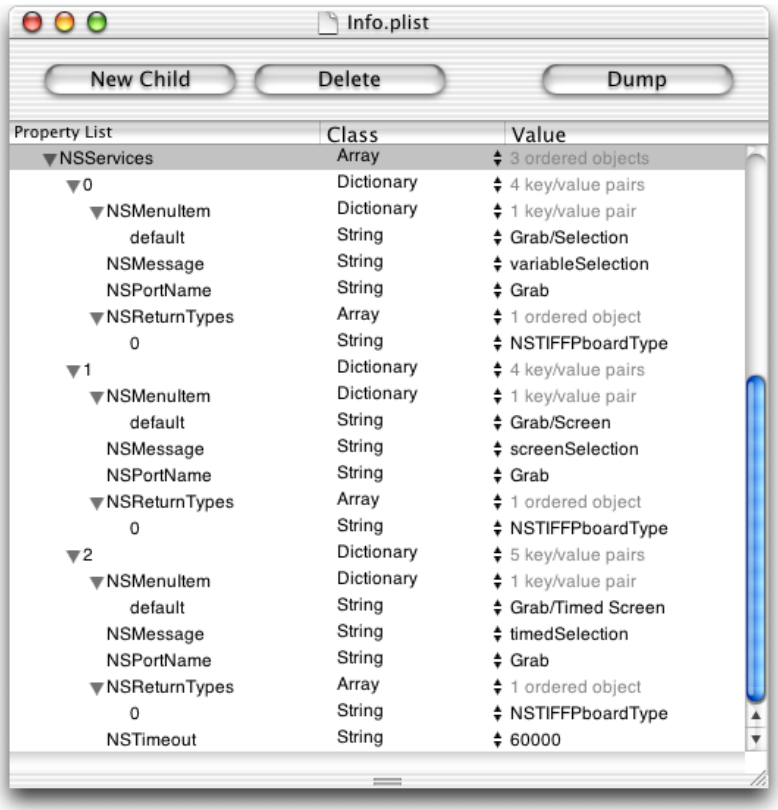
The `NSServices` property has three entries, one for each service offered by Grab. The first entry is for the menu item Grab > Selection. The slash notation—`Grab/Selection`—specifies that Selection should be an item in the Grab submenu. (See Figure 4 (page 11).)

Note that for each of the three entries, the port name is Grab. As mentioned, the port name is usually the application name.

Each entry has one return type, `NSTIFFPboardType`. An application could have more than one return type per entry, and the return types don't necessarily need to be the same for each entry.

The entry for `Grab/Timed Screen` is the only entry that has a specified timeout value. This optional entry is needed in this case so that the Grab application can wait for the user to set up the screen before taking a screen shot.

Figure 1 The NSServices property for the Grab application



Providing a Service

Suppose you are working on a program to read USENET news, and have an object with a method to encrypt and decrypt articles, such as the one in Listing 1 (page 19). News articles containing offensive material are often encrypted with this algorithm, called "rot13," in which letters are shifted halfway through the alphabet.

Listing 1 Text encryption method

```
- (NSString *)rotateLettersInString:(NSString *)aString
{
    NSString *newString;
    unsigned length;
    unichar *buf;
    unsigned i;

    length = [aString length];
    buf = malloc( (length + 1) * sizeof(unichar) );
    [aString getCharacters:buf];
    buf[length] = (unichar)0; // not really needed....
    for (i = 0; i < length; i++) {
        if (buf[i] >= (unichar)'a' && buf[i] <= (unichar) 'z') {
            buf[i] += 13;
            if (buf[i] > 'z') buf[i] -= 26;
        } else if (buf[i] >= (unichar)'A' && buf[i] <= (unichar) 'Z') {
            buf[i] += 13;
            if (buf[i] > 'Z') buf[i] -= 26;
        }
    }
    newString = [NSString stringWithCharacters:buf length:length];
    free(buf);
    return newString;
}
```

Implementing the Service Method

Since this feature is generally useful as a simple encryption scheme, it can be exported to other applications. To offer this functionality as a service, write a method such as the one in Listing 2 (page 19).

Listing 2 Service method

```
- (void)simpleEncrypt:(NSPasteboard *)pboard
            userData:(NSString *)userData
            error:(NSString **)error
{
    NSString *pboardString;
    NSString *newString;
    NSArray *types;
```

```
    types = [pboard types];
    if (![types containsObject:NSStringPboardType]) {
        *error = NSLocalizedString(@"Error: couldn't encrypt text.",
                        @"pboard couldn't give string.");
        return;
    }
    pboardString = [pboard stringForType:NSStringPboardType];
    if (!pboardString) {
        *error = NSLocalizedString(@"Error: couldn't encrypt text.",
                        @"pboard couldn't give string.");
        return;
    }
    newString = [self rotateLettersInString:pboardString];
    if (!newString) {
        *error = NSLocalizedString(@"Error: couldn't encrypt text.",
                        @"self couldn't rotate letters.");
        return;
    }
    types = [NSArray arrayWithObject:NSStringPboardType];
    [pboard declareTypes:types owner:nil];
    [pboard setString:newString forType:NSStringPboardType];
    return;
}
```

The method providing the service is of the form *messageName*`:userData:error:` and takes arguments as shown in Listing 2 (page 19). The method itself takes data from the pasteboard as needed, operates on it, and writes any results back to the pasteboard. In case of an error, the method simply sets the pointer given by the error argument to a non-nil NSString and returns. The error message is logged to the console. The *userData* argument is not used here.

If implementing the service in Java, the invoked method has the form

```
String messageName(NSPasteboard pboard, String userData)
```

This method returns `null` if successful; otherwise, it returns the error message string.

Registering the Service Provider

Now you have an object with methods that allow it to perform a service for another application. Next, you need to register the object at run time so the services facility knows which object to have perform the service. You create and register your object in the `applicationDidFinishLaunching:` application delegate method (or equivalent) with NSApplication's `setServicesProvider:` method. If your object is called `encryptor` you create and register it with this code fragment:

```
EncryptoClass *encryptor;
encryptor = [[EncryptoClass alloc] init];
[NSApp setServicesProvider:encryptor];
```

If you are writing a plain Foundation tool, which lacks an NSApplication object, register the service object with the `NSRegisterServicesProvider` function. Its declaration is:

```
void NSRegisterServicesProvider(id provider, NSString *portName)
```

provider is the object that provides the services, and *portName* is the same value you specify for the NSPortName entry in the services specification. After making this function call, you must enter the run loop to respond to service requests.

You can register only one service provider per application. If you have more than one service to provide, a single object must provide the interface to all of the services.

Service requests can arrive immediately after registering the object, in some circumstances even before exiting applicationDidFinishLaunching:. Therefore, register your service provider only when you are completely ready to process requests.

Advertising the Service

For the system to know that your application provides a service, you must advertise that fact. You do this by adding an entry to your application project's Info.plist file. The entry you add is called the service specification. In our example, the service specification looks like this:

```
NSServices = (
    {
        NSPortName = NewsReader;
        NSMessage = simpleEncrypt;
        NSSendTypes = (NSStringPboardType);
        NSReturnTypes = (NSStringPboardType);
        NSMenuItem = {
            default = "Encrypt Text";
        };
        NSKeyEquivalent = {
            default = E;
        };
    }
);
```

Installing the Service

A service can be offered as part of an application, such as Mail, or as a standalone service—one without a user interface that is intended for use only in the Services menu. Applications that offer services should be built with the .app extension and installed in the Applications folder (or a subfolder) in one of the four file-system domains—System, Network, Local, and User. (See "File-System Domains" in *System Overview* for details.) A standalone service should be built with the .service extension and stored in the Library/Services folder in one of these domains.

The list of available services on the computer is built each time a user logs in. After installing your service in either an Applications or Library/Services directory, you need to log out and back in again before the service becomes available. You can force an update of the list of services without logging out by calling the following function:

```
void NSUpdateDynamicServices(void)
```

Running applications are not affected, but applications launched afterwards get the new list of services.

Using Services

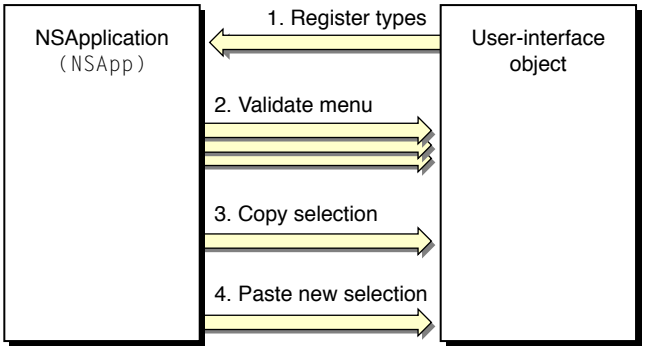
The default nib file created for new Cocoa applications contains a Services menu in the application menu, so there is nothing else you need to do for your application to work with the services facility; your application automatically has access to all appropriate services provided by other applications. If you need to construct menus programmatically, you simply designate the NSMenu that you want as your Services menu with NSApplication's `setServicesMenu:` method.

If you subclass NSView or NSWindow (or any other subclass of NSResponder), you need to implement it such that it interacts properly with the services facility. Tying custom NSViews or NSWindows into the services facility falls into the following steps:

1. Registering your user-interface objects for services
2. Validating the Services menu items for the current selection
3. Sending the current selection to the service
4. Receiving data from the service to replace the current selection

These steps are illustrated in Figure 1 (page 23).

Figure 1 Using services



When a pure Service provider is invoked (in other words, no send types), step 3 is skipped. When a pure Service processor is invoked (in other words, no return types), step 4 is skipped.

The following sections cover each of these steps. A final section, "Invoking a Service Programmatically" (page 26), shows how to invoke a service in your code.

Registering Objects for Services

The Services menu does not contain every service offered by other applications. For example, in a text editor a service to invert a bitmapped image is of no use and should not be offered. Which services appear in the Services menu is determined by the data types that the objects in the application—specifically the NSResponder objects—can send and receive through the pasteboard.

An NSResponder registers these data types using the NSApplication Objective-C method `registerServicesMenuSendTypes:returnTypes:` or Java method `registerServicesMenuTypes`. Application Kit objects already do this for the basic text services, but your custom NSResponder subclass must do this to expand the list. A convenient location is in your subclass's `initialize` class method, which is guaranteed to be invoked by the runtime before any other method of the class. All types used by instances of the class must be registered, even if they are not always available; Services menu items are enabled and disabled dynamically based on what is available at the moment, as described in "Validating Services Menu Items" (page 24).

An object does not have to register the same types for both sending and receiving. Suppose you are writing a rich text editor that can send unformatted and rich text, but can only receive unformatted text. Here is a portion of the initialization method for a text-editor's NSView subclass:

```
+ (void)initialize
{
    static BOOL initialized = NO;
    /* Make sure code only gets executed once. */
    if (initialized == YES) return;
    initialized = YES;

    sendTypes = [NSArray arrayWithObjects:NSStringPboardType,
                    NSRTFPboardType, nil];
    returnTypes = [NSArray arrayWithObjects:NSStringPboardType,
                    nil];
    [NSApp registerServicesMenuSendTypes:sendTypes
                    returnTypes:returnTypes];
    return;
}
```

Your NSResponder object can register any pasteboard data type, public or proprietary, common or rare. If it handles the public and common types, of course, it has access to more services. See the NSPasteboard class specification for a list of standard pasteboard data types.

Validating Services Menu Items

While your application is running, various types of data can be selected and available for transfer on the pasteboard. If a service does not apply to the type of the selected data, its menu item needs to be disabled. To check whether a service applies, the application object sends `validRequestorForSendType:returnType:` messages to Objective-C objects, and `validRequestorForTypes` to Java objects, in the responder chain to see whether they have data of the type used by that service. While the Services menu is visible, this method is invoked frequently—typically many times per event—to ensure that the menu items for all service providers are properly enabled: It is

sent for each combination of send and return types supported by each service and possibly for many objects in the responder chain. Because this method is invoked so frequently, it must be fast so that event handling does not fall behind the user's actions.

The following example shows how this method can be implemented for an object that handles unformatted text:

```
- (id)validRequestorForSendType:(NSString *)sendType
            returnType:(NSString *)returnType
{
    if ( (!sendType || [sendType isEqual:NSStringPboardType]) &&
        (!returnType || [returnType isEqual:NSStringPboardType]) ) {
        if ( (!sendType || [self selection]) &&
            (!returnType || [self isEditable]) ) {
            return self;
        }
    }
    return [super validRequestorForSendType:sendType
            returnType:returnType];
}
```

This implementation checks both the types indicated and the state of the object. The object is a valid requestor if the send and return types are unformatted text or simply are not specified, and if the object has a selection and is editable (when send and return types are given). If this object cannot handle the service request in its current state, it invokes its superclass' implementation.

`validRequestorForSendType:returnType:` is sent along an abridged responder chain, comprising only the responder chain for the key window and the application object. The main window is excluded.

Sending Data to the Service

When the user chooses a Services menu command, the responder chain is checked with `validRequestorForSendType:returnType:` and the first object that returns a value other than `nil` is called upon to handle the service request by providing data (if any is required) with a `writeSelectionToPasteboard:types:` message. Java objects are sent a `writeSelectionToPasteboardOfTypes` message. You can implement this method to provide the data immediately or to provide the data only when it is actually requested. Here is an implementation for an object that writes unformatted text immediately:

```
- (BOOL)writeSelectionToPasteboard:(NSPasteboard *)pboard
types:(NSArray *)types
{
    NSArray *typesDeclared;

    if ([types containsObject:NSStringPboardType] == NO) {
        return NO;
    }
    typesDeclared = [NSArray arrayWithObject:NSStringPboardType];
    [pboard declareTypes:typesDeclared owner:nil];
    return [pboard setString:[self selection]
                    forType:NSStringPboardType];
}
```

This method returns `YES` if it successfully writes or declares any data and `NO` if it fails. If you have large amounts of data or you can provide the data in many formats, you should provide the data only on demand. You declare the available types as above, but with an owner object that responds to `pasteboard:provideDataForType:`. See the NSPasteboard class specification for more details.

Receiving Data from the Service

Once the service requestor writes data to the pasteboard, it waits for a response as the service provider is invoked to perform the operation; if the service does not return data, of course, the requesting application simply continues running and none of the following applies. The service provider reads the data from the pasteboard, works on it, and then returns the result. At this point the service requestor is sent a `readSelectionFromPasteboard:` message telling it to replace the selection with whatever data came back. (The Java method has the same name.) The simple text object can implement this method as follows:

```
- (BOOL)readSelectionFromPasteboard:(NSPasteboard *)pboard
{
    NSArray *types;
    NSString *theText;

    types = [pboard types];
    if ( [types containsObject:NSStringPboardType] == NO ) {
        return NO;
    }
    theText = [pboard stringForType:NSStringPboardType];
    [self replaceSelectionWithString:theText];
    return YES;
}
```

This method returns `YES` if it successfully reads the data from the pasteboard, `NO` otherwise.

Invoking a Service Programmatically

Though the user typically invokes a standard service by choosing an item in the Services menu, you can invoke it in code using this function:

```
BOOL NSPerformService( NSString *serviceItem, NSPasteboard *pboard )
```

This function returns `YES` if the service is successfully performed, `NO` otherwise. *`serviceItem`* is the name of a Services menu item (in any language). It must be the full name of the service, including the submenu and slash; for example, "`Mail/Selection`". *`pboard`* contains the data to be used for the service, and when the function returns contains the data resulting from the service. You can then do with the data what you wish.

Creating the NSServices Property

Applications must use their information property list to advertise the services they provide. You need to add the `NSServices` property, and it must have one dictionary entry for each service you provide. Each dictionary entry must have these keys: `NSMessage`, `NSPortName`, and `NSMenuItem`. Each entry must have one or both of `NSSendTypes` and `NSReturnTypes`. Each entry can optionally have these keys: `NSKeyEquivalent`, `NSUserData`, and `NSTimeout`. See "Services Properties" (page 15) for a description of each key and examples of the `NSServices` property in an information property list.

To add services properties to the information property list for a service application or a standalone service do the following:

1. Open your services application or standalone services project in Xcode.
2. Click the Targets tab, then click the appropriate target in the Targets list.
3. Click the Application Settings tab, then click Expert.
4. Click the New Sibling button.
5. Type `NSServices` in the Property List column.
6. Choose Array from the Class pop-up menu.
7. Click the disclosure triangle next to `NSServices`, then click the New Child button.

   When you click the disclosure triangle, the New Sibling button changes to New Child.
8. Set the array element's class to Dictionary.
9. Click the disclosure triangle next to the array element, then click the New Child button.
10. Type an `NSServices` keyword in the Property List column, make sure its class is set appropriately, then type or choose a value.

    Click New Sibling to add the other required keywords to this array element.

    See "Services Properties" (page 15) for a discussion of keywords and their classes.

You need to add an array element to the `NSServices` property for each service your application provides. To add another array element, click `NSServices`, click the New Child button, then follow steps 8 through 10.

Document Revision History

This table describes the changes to *System Services*.

Date	Notes
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.